# Query Processing Techniques for Multiversion Access Methods

Jochen van den Bercken Bernhard Seeger

Fachgebiet Informatik, University of Marburg
Hans-Meerwein-Str., D-35032 Marburg, Germany
e-mail: {bercken, seeger}@informatik.uni-marburg.de

## Abstract

Multiversion access methods have been emerged in the literature primarily to support queries on a transaction-time database where records are never physically deleted. For a popular class of efficient methods (including the multiversion B-tree), data records and index entries are occasionally duplicated to separate data according to time. In this paper, we present techniques for improving query processing in multiversion access methods. In particular, we address the problem of avoiding duplicates in the response sets. We first discuss traditional approaches that eliminate duplicates using hashing and sorting. Next, we propose two new algorithms for avoiding duplicates without using additional data structures. The one performs queries in a depth-first order starting from a root, whereas the other exploits links between data pages. These methods are discussed in full details and their main properties are identified. Preliminary performance results confirm the advantages of these methods in comparison to traditional ones according to CPU-time, disk accesses and storage.

**Keywords:**
temporal databases, query processing, multiversion access methods, optimization and tuning techniques, physical database design

## 1 Introduction

Data base management systems (DBMS) have primarily been developed to protect the current state of a database against errors and loss, whereas only a few systems (e.g. POSTGRES [Sto 87]) are also able to preserve past states. The past states of a database are however very important for various applications like land register systems [BFA 91], decision support systems and on-line analytical processing (OLAP) [CCS 93]. A primary task of a land register system, for example, is to keep track of the ownership of real-estate over time. For these applications, queries should be supported on any state, past or current, and on multiple states (e.g. to show the temporal development of data). This type of database is called a transaction-time database [SA 85].

Traditional query processing techniques and access methods are not suitable to manage transaction-time databases for the following reasons. First, since records are never physically deleted transaction-time databases are extremely large in size. In general, a transaction database cannot completely be kept on magnetic disk [LS 89], but some parts have to be stored on tertiary storage (e.g. optical disks or tapes). Second, queries are more complex because they can refer to the key and time dimension. An important type of query is the *range-period query*: Given a key range R and a time period P, find the records whose keys are in R during P. Special cases of this query are the *key-period query* and the *range-time query* where the key range and the time period is reduced to a single key and time instant, respectively. In a land register system, the query „Find all owners of parcels which are in a given rectangle over the last 10 years" is an example of a range-period query. Another requirement might be that the output of a range-period query has to be sorted according to time. Although various special access methods have been developed for transaction-time databases, see [ST 94] for an excellent survey, the authors are not aware of a discussion on processing range-period queries. Previous studies are primarily related to supporting either key-period queries or range-time queries.

In this paper, we present a detailed discussion of various algorithms for processing range-period queries in transaction databases using special access methods. In the following, we refer to these methods as *multiversion access methods* (MVAMs). Our discussion is primarily based on the multiversion B-tree (MVBT) [BGO+ 93] and other methods ([Eas 86], [LS 89], [LS 90], [MKW 96]) closely related to the MVBT. The MVBT optimally solves the range-time query problem [BGO+ 93] and therefore, is also an ideal candidate for processing the more general range-period query. The MVBT achieves its performance by duplicating data records and index entries. Range-period queries which require access to multiple versions however suffer from the duplicates in various ways. First of all, when traditional algorithms for processing range que-



**Proceedings of the 22nd VLDB Conference**
**Mumbai(Bombay), India, 1996**

ries are used to evaluate range-period queries, it will be not distinguished between ordinary records and their duplicates. Consequently, these algorithms would allow duplicates in the response set. If duplicates are not desired, additional effort is required to remove them from the response set. Crucial to efficiency is also the fact that duplicates of directory entries exist and that the traditional query processing algorithm would also not identify these duplicates. Consequently, the same page can be visited more than once. In the following, we present various algorithms for evaluating range-period queries without reporting duplicates. Although the algorithms are presented in the context of the MVBT, they are directly applicable to other multiversion access methods (like the write-once B-tree [Eas 86] and the time-split B-tree [LS 89]).

To the best of the authors knowledge, the problem of duplicate avoidance has not been addressed in the context of multiversion access methods so far. The problem is not only restricted to transaction-time databases, but also occurs in spatial data bases. For example, spatial access methods like the $R^+$-tree [SRF 87] also produce duplicates (only records) which have to be eliminated from the response set of a query in a second step. Our techniques for avoiding duplicates (developed for the MVBT) can also be exploited for avoiding the duplicates while processing spatial queries on $R^+$-trees and related spatial access methods.

Section 2 of the paper introduce to the problem of organizing a transaction-time database. Special attention is given to the foundations of multiversion access methods. Section 3 describes four algorithms designed for performing range-period queries with duplicate avoidance. We first discuss traditional techniques (hashing and sorting). Next, we present a new depth-first traversal algorithm that avoids duplicates by testing so-called reference points. Our fourth algorithm generalizes the link traversal known from the $B^+$-tree. Section 4 shows the results of a set of experiments. The conclusions are in section 5.

# 2 Preliminaries

In this section, we introduce to transaction-time databases and present a brief review of multiversion access methods (*MVAMs*) designed for organizing records of a transaction-time database. Special attention is paid to the multiversion B-tree (*MVBT*) which serves as the underlying MVAM throughout the paper.

## 2.1 Transaction-Time Databases

A *transaction-time database* consists of records which belong to different versions (states) of a file. For sake of simplicity we assume that the versions of a file are numbered contiguously starting from 0. The version with the highest number is called the *current* version. A record which belongs to the current version of a file is said to be *live*, and *dead* otherwise. A change (insertion, deletion, update) can only be applied to the current version, whereas queries are supported on any version, current or past. Moreover, we also consider queries which require access to multiple versions. Each change creates a new version; the i-th change is applied to version i-1 at time $t_i$ and creates version i. A *record of a transaction-time database* can be represented as a tuple consisting of a key, a time interval and some associated information. The left and right point of the time interval represent the time when the record was inserted into and (logically) deleted from the transaction-time database, respectively. In the following, the interval is called *life-span* of a record. The right bound of the life-span can adopt the special value *now* that identifies a record to be live.

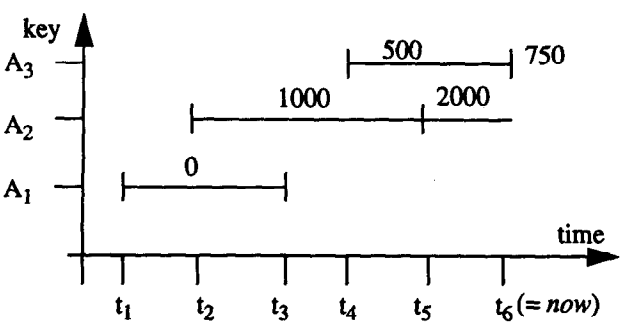



Figure 1: An example of a transaction-time database

An example of a transaction-time database is depicted in Figure 1. Records are illustrated as intervals in a two-dimensional data space. The information part of an record is the number reported above the interval. Records with keys $A_1$ and $A_2$ are inserted at time $t_1$ and $t_2$, respectively. At time $t_3$, the record with key $A_1$ is (logically) deleted. A record with key $A_3$ is inserted at $t_4$. An update on the record with key $A_2$ creates version 5. Another update on key $A_3$ creates version 6 which is the current version of the transaction-time database.

## 2.2 Multiversion Access Methods with Data Duplication

In this subsection, we present the fundamentals of MVAMs that use the technique of record duplication. For sake of concreteness, we base our discussion on MVAMs derived from $B^+$-trees.

MVAMs organizes a dynamic set of intervals in a two-dimensional dataspace, see Figure 1. Their approach to mapping intervals to pages is closely related to spatial access methods that use the clipping technique [SK 88]. The basic idea is to partition the dataspace into rectilinear two-dimensional rectangles called *page regions*. Each page region is associated with exactly one page such that, when an interval intersects with the page region, it will be stored in the corresponding page. Note that this results in storing the same interval in different pages. The remaining problem is then how to organize the page regions. This is solved by recursive partitioning the data space into page regions (until the number of page regions fits in a page). The process of recursive partitioning is recorded in a balanced tree where the nodes are associated with pages. We use the

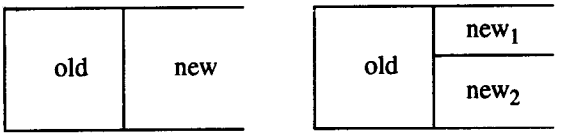
terms *data pages* and *directory pages* for leaves and non-leaves, respectively. This data organization is closely related to $R^+$-trees [SRF 87]. In contrast to $R^+$-trees, however, MVAMs also allow the occurrence of the same index entry in different directory pages.

An insertion of a new interval or an update of an existing interval can result in an overflow of a page. Then, a structural change has to be performed on the MVAM. There are key-splits, version-splits and combinations of both.

A *key-split* is similar to a split in an ordinary $B^+$-tree. A separator key is used for splitting the page region of the overflowing page into two. Records below of the separator key remain in the original page, whereas the other records are moved into a new page.

For a *version-split*, a time instant is selected for separating time-varying records. In contrast to a key-split, some of the records may intersect with both page regions of the newly created pages and therefore, they also have to be stored in both pages. A special property of transaction-time databases is that a split is only performed on current pages, i.e. pages whose page regions contain the current version. Thus, a version-split creates a historical page and a current page, see Figure 2 a). When the current time (*now*) is used for the version-split, the number of records stored in the (new) current page will be minimal among all possible version-splits. Consequently, the page will not be involved in a split for a rather long time period. Note that a historical page will not be modified any more.

When a version-split creates an almost full (current) page, only a few updates and insertions would be sufficient to trigger the next split on the same page. In order to prevent such frequently occurring split operations, a key-split can be performed on such page immediately after the version-split, see Figure 2 b).




a) version-split b) version-split & key-split

Figure 2: Structural changes by split operations

For each data page resulting from a split, a *directory entry* is inserted into the parent page like it is known from $B^+$-trees. In addition to the pointer to the page, the entry consists of the page region. An overflow of a directory page can again be eliminated by using a key-split, a version-split or a combination of both. Note that a version-split of a directory page copies current entries to a new page, i.e. different copies of the same directory entry exists in a MVAM. Therefore, a MVAM derived from a $B^+$-tree is actually not a tree, but a directed acyclic graph.

The write-once B-tree (WOBT) [Eas 86] and the time-split B-tree (TSBT) [LS 89], [LS 90] are prime examples of MVAMs which organize a transaction-time database only using key-splits, version-splits and combinations of both. Both methods do not efficiently support (logical) deletions of records. To the contrary, it is assumed that the number of records grows constantly with an increasing number of versions. The TSBT uses a more sophisticated splitting policy resulting in a lower degree of redundancy *for data records* compared to the WOBT. However, the TSBT cannot efficiently support range-time queries on past versions since records close in key space at some version are not stored in the same data page. The goal for the design of the MVBT was to overcome these drawbacks of previous methods.

### 2.3 Review of the Multiversion B-Tree

The multiversion B-tree (MVBT) is a MVAM that efficiently supports (logical) deletion of records. The MVBT offers asymptotically optimal worst-case performance for insertions, deletions and updates as well as for range-time queries. An important feature of the MVBT is to guarantee a minimum *key density* for every page and for every version represented in the page, i.e. a page whose page region covers a version $v$ contains at least $d$ records/entries which belong to version $v$. It is important to (worst-case) efficiency that $d$ is a linear function in $b$ (the capacity of a page).

A MVAM that allows deletion like the MVBT requires a merging policy to guarantee a minimum key density for all versions in a page. If a *weak version underflow* occurs after a deletion, i.e. the key density of a current page drops below a threshold, a sibling page is determined (according to key). Then, a version-split is performed on both pages using the current time and the resulting pages are merged into one, see Figure 3 a). This merging policy for eliminating a weak version underflow can be used for data pages as well as directory pages.

The MVBT also prevents that long sequences of successive split and merge operations are performed on a single page. This is achieved by fulfilling the following requirement for the current pages: Immediately after a current page was created the number of records/entries must be between a lower bound *low* and an upper bound *up*. It is required that *low* is greater than $d$ (the key density). Let $b$ be the capacity of a page. For sake of simplicity, we consider the following setting of the parameters throughout the paper: $d = b/5$, $low = 2b/5$, $up = 4b/5$. Moreover, we assume that $b$ is a multiple of 5. If the occupancy of a newly created current page is greater than up ($=4b/5$) it will be split into two using a key-split. If the occupancy is less than low ($=2b/5$) a merge is performed using an appropriate sibling page. This can result in a page whose occupancy can again be greater than *up* and therefore, another key-split is required to meet the requirements of the MVBT, see Figure 3 b).

Overall, there are four different types of structural changes: version-split, version-split & key-split, merge and merge & key-split. An example depicted in Figure 4 illustrates the four different cases. The partitioning of the MVBT contains 11 page regions which belong to the data pages. A version-split was performed on page C at time $t_2$. At time $t_3$, a combination of version split and key split created

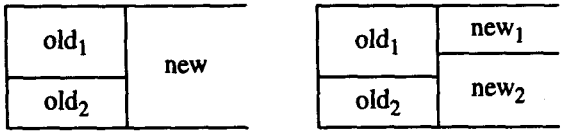



Figure 3: Structural changes by merge operations

pages E and F. Pages F and G were merged together at time $t_5$. A combination of merge and key-split was performed on pages I and H at time $t_6$.

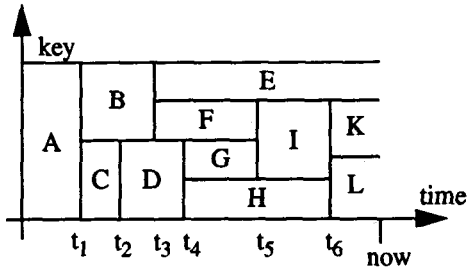



Figure 4: Example of a partitioning of the MVBT

Another unique feature of the MVBT (different to the TSBT and WOBT) is that it contains various root pages valid for different time periods. Note that there is only one current root. Consider for example that a version-split is performed on the (current) root P. Then, the live entries of P are copied into a new root page and P is marked to be dead. The references to the different roots are stored in another data structure called root*. One of the advantages of allowing different roots is that the time complexity of update operations only depends on the number of records stored in the current version and not on the total number of versions (as it can be observed for the WOBT and TSBT). In general, the size of root* is rather small such that it can almost always be kept resident in main memory.

The advantages of the MVBT are not given for free. Although the degree of redundancy is bound by a *constant* in the worst-case, the MVBT uses more space than the WOBT and the TSBT. For MVAMs with data duplication, there is a trade-off between query time and disk space.

## 3 Algorithms for Processing Range-Period Queries

In this section, we examine several algorithms how multi-version access methods can be used for the evaluation of range-period queries. We primarily restrict our discussion to tree-structured access methods with data duplication. The range-period query is defined in the following way:

> Given a key range R and a time period P, find the records whose keys are in R during P.

The naive solution to evaluating range-period queries is to traverse the tree in depth-first order from the root down to the leaves. This strategy is well-known from processing range queries in hierarchical multidimensional access methods (e.g. R-trees, K-D-B-trees). The depth-first traversal strategy has the advantage that only a path of the tree has to be kept in main memory while processing a query. For MVAMs, the naive algorithm however leads us to the following two serious problems. First of all, the response set can contain duplicates. Such duplicates should be avoided in many cases, e.g. when the „distinct" qualifier occurs in a SQL statement or when a range-period query delivers the answers to the next operator of an operator tree (e.g. an aggregation like *sum* or *avg*). Second, the same directory entry can be accessed more than once while a query is evaluated. Consequently, pages are accessed more than once. Consider that a page $P$ is visited $x$ times, $x > 0$. Then, all qualifying entries/records in the subtree of $P$ are also visited at least $x$ times. This process can be repeated on each level of the tree. As a consequence, the degree of redundancy has an exponential influence (in the height of the tree) on the number of redundant records qualifying the query predicate. This effect is illustrated in Figure 5 where the gray-colored rectangles illustrate the pages of the MVBT visited while processing a range-period query. The naive algorithm would visit pages A, B and E once, pages C, D, G twice, page H thrice and page F four times.

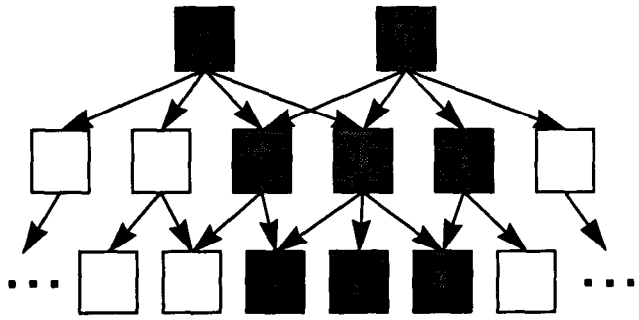


Figure 5: Visited pages of a MVBT

In the following, we present four algorithms for evaluating range-period queries without producing duplicates. The first two approaches use classical techniques such as hashing and sorting. In the second subsection, we present a new approach to detecting duplicates using the so-called reference point. This approach is based on a depth-first traversal of the tree. For each qualifying entry, the reference point is computed and tested against a certain condition. If so, the algorithm follows the corresponding reference down to the next node. In the third subsection, we present an algorithm similar to the one for performing range queries on $B^+$-trees. Data pages are connected through links which are additionally stored in the pages. In contrast to $B^+$-trees, however, data pages may have more than one predecessor page. This algorithm can be used when the output has to be sorted with respect to time. Moreover, the pages can be read in the order of their creation which is important to efficiency in particular when the data is stored on tertiary storage.

For sake of concreteness, we base our presentation of query processing techniques on the MVBT. Our algorithms are described in a such a way that they can directly be applied to the TSBT and similar MVAMs.

### 3.1 Traditional Approaches

In this subsection, we present two well-known techniques for avoiding duplicates in the response set.

#### 3.1.1 Hashing

Hashing is a widely used technique to eliminate duplicates and to perform other relational operations [Bra 84]. For processing a range-period query on the MVBT, a hash table $HT_{data}$ is provided for collecting the answers. The algorithm starts in the root of the tree and traverses down to the leaves in depth-first order. For a qualifying directory entry, the search is always directed to the corresponding page of the entry, whereas for a qualifying record, hash table $HT_{data}$ is first investigated whether it does contain the record or not. If the record is not in $HT_{data}$, it is accepted as an answer. In order to avoid duplicates, the record is then inserted into $HT_{data}$. In the other case (the record is in $HT_{data}$), the answer was already reported earlier. In the following, we use the shortcut $DF_{Hash}$ to refer to the algorithm.

First of all, $DF_{Hash}$ obviously guarantees that answers occur only once in the response set. However, it also offers several drawbacks. First, the storage overhead for the hash table is linear in the *number of answers*. Since transaction-time databases are very large, the response set of a query can be larger than the available main memory. Then, the hash table has to be paged out on disk. Second, this method does not detect redundant directory entries. Therefore, the same page can be visited more than once. This will increase the CPU-cost because the buffer has to be examined multiple times for reading records/entries from the same page. Moreover, the average number of accesses to a record will also increase (see the example of Figure 5) and therefore, the hash table has to be probed rather frequently.

In order to detect redundant directory entries, we can slightly modify $DF_{Hash}$ using a second hash table $HT_{dir}$ for collecting qualifying directory entries. When a directory page is examined, we first test each entry whether it is in $HT_{dir}$. If so, the corresponding page was already visited previously. Otherwise, the entry is inserted into $HT_{dir}$ and the search proceeds down to the corresponding page. This algorithm guarantees that redundant entries are recognized and that a page is visited only once. Thus, it reduces the number of accesses to the hash table, but it does not reduce the storage overhead of $DF_{Hash}$.

#### 3.1.2 Sorting

The other classical technique for duplicate elimination is sorting. A range-period query is performed in three phases. In the first phase, the tree is traversed in depth-first order and the qualifying records are copied into a buffer. In the second phase, the buffer is sorted (for example according to the insertion time of the entries). Thereafter, in the third phase, a linear scan through the buffer is performed to remove the duplicates. This algorithm is called $DF_{Sort}$.

A drawback of $DF_{Sort}$ is that the required buffer space growth linear in the number of both answers and duplicates. Therefore, the storage overhead is higher than the one of $DF_{Hash}$. Another disadvantage of sorting is that it is not very suitable for pipelining queries.

Overall, both algorithms $DF_{Hash}$ and $DF_{Sort}$ have a high storage consumption for performing a query. In the next subsection, we present methods with considerably lower storage consumption.

### 3.2 A Depth-First Algorithm with Duplicate Avoidance

In this section we propose a new algorithm called $DF_{Ref}$ for processing range-period queries that does not require additional data structures for detecting both redundant qualifying entries and redundant answers. The implementation of the method is simple and the computational overhead is very low. In order to illustrate the basic idea of our approach, let us first discuss an example illustrated in Figure 6.

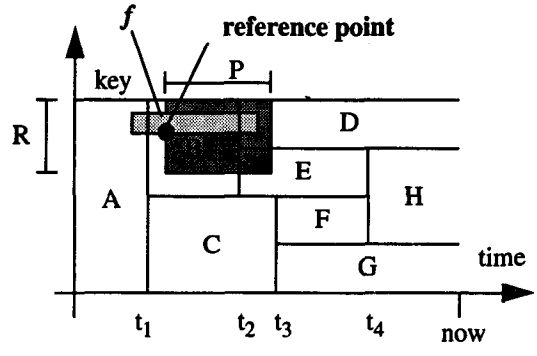



Figure 6: Computation of the reference point

In our example, we consider a MVBT with 8 directory pages labeled A,...,H. The bright gray rectangle illustrates an entry (with label *f*) stored on the next lower level. A copy of entry *f* is stored in each of the pages A, B and D since it was live at time $t_1$ and $t_2$. Now consider a key range R and a time period P and let us process the corresponding range-period query as depicted in Figure 6. The range-period query has to visit pages B, D and E since the query rectangle intersects with their page regions. For each qualifying entry, we compute the intersection between the query rectangle and the rectangle of *f*. The lower left corner of the intersection is called the *reference point* of entry *f*. Since there are only copies of entry *f* in the different pages, we compute the same reference point for each occurrence of *f*. Thereafter, we check whether the reference point is in the page region. In our example, the page region of B contains the reference point of *f*, whereas the other page regions do not. Therefore, the copy of entry *f* in page B is selected for proceeding the search down to the page stored

on the next lower level in the tree. A more formal description of the algorithm follows.

```
1 Algorithm DF_Ref( Entry f; key_range [R_L ,R_U);
                    time_period [P_B,P_E))
2  N = GetPage(f);
3  if (N is a directory page)
4     compute the page region [K_L,K_U) x [T_L,dummy);
      /* this information is stored in f */
5     for (each entry g in N) do
6        compute the key range [low, up) of g;
         /* low < up */
7        compute the life-span [t_B,t_E) of g;
8        if ( [low, up) ∩ [R_L, R_U) ≠ ∅ and
             [t_B, t_E) ∩ [P_B, P_E) ≠ ∅ )
9           (ref_K,ref_T) = (max(low, R_L), max(t_B, P_B));
            /* (ref_K,ref_T) is the reference point */
10          if ((K_L ≤ ref_K < K_U) and (T_L ≤ ref_T))
11             DF_Ref(g, [R_L ,R_U), [P_B,P_E));
12 else /* N is a data page */
13    ... /* this code is very similar to the case of N
             being a directory page */
14 end; /* DF_Ref */
```

In addition to key-range R and time period P, the entry referring to the root of the tree is used as an input parameter when algorithm $DF_{Ref}$ is initially called. In case of the MVBT, such an entry is found in root*. The algorithm first retrieves the corresponding page and then distinguishes between data pages and directory pages. For the latter case, our description goes into details. Next, we partly compute the page region of page *N*. The relevant parts of the page region can easily be derived from the entry *f* which is referring to *N*. Note that we do not require the time when *N* was (logically) deleted. For each of the entries of page *N*, we compute its key range and its life-span. Thereafter, the entry is tested whether its key range and life-span intersects with *R* and *P*, respectively. If so, the reference point is computed and is checked whether it does belong to the page region or does not. If so, a recursive call to the algorithm will follow.

The naive depth-first traversal algorithm is almost equal to $DF_{Ref}$ except for three additional lines (line 4,9 and 10). The computational overhead of our algorithm consists of computing the reference point (line 9) and testing whether it is in the page region of N or not (line 10). The following theorem shows that algorithm $DF_{Ref}$ gives the desired behavior.

**Theorem 1:** The algorithm $DF_{Ref}$ has the following properties:

- a qualifying page is visited only once
- an answer is reported only once.

Due to space limitations we omitted the proof of Theorem 1. The interested reader is referred to [BS96].

An advantage of $DF_{Ref}$ is that no additional memory and data structure is required to detect duplicates. Note that in contrast to $DF_{Ref}$ the algorithms based on hashing and sorting additionally require memory to keep the hash table and buffer resident, respectively. The traversal strategy is only based on local information stored in the page and in the parent page. Therefore, we can also use this algorithm when the MVBT would be distributed among different computing nodes [MKW 96] without causing communication for eliminating duplicates.

Let us emphasize that the reference-point method is not restricted to the MVBT, but is also applicable to the WOBT, TSBT and similar methods. Moreover, the same multiversion technique that modifies a B-tree to a multiversion B-tree can also be applied to R-trees [Gut 84] and other multidimensional access methods. The depth-first algorithm combined with the reference point method would be still usable without modifications for such a multiversion R-tree.

The reference-point method avoids access to pages which are referenced by duplicated directory entries. Thus, qualifying pages are investigated only once. Therefore, the number of disk accesses will be considerably less in comparison to a pure depth-first algorithm without using the reference-point method. In contrast to $DF_{Hash}$ and $DF_{Sort}$, this is achieved without using any additional data structures.

A drawback of the depth-first algorithm is still that all qualifying directory entries including the duplicated ones have to be visited. Thus, the overhead of traversing the directory pages can be rather high, in particular for small queries. The question therefore arises how the cost for processing directory pages can be reduced.

### 3.3 A Link Algorithm with Duplicate Avoidance

In this section, we present a new algorithm called $Link_{Ref}$ for processing range-period queries. Algorithm $Link_{Ref}$ exploits links between the data pages of the MVAM. This approach is restricted to a certain class of MVAMs including the ones derived from $B^+$-trees. Although there are some similarities to the traditional algorithm for processing range queries on $B^+$-trees, the link approach for the MVBT shows some unique features. For example, the data pages of the MVBT cannot be linked together in a linear list (as it is known from the $B^+$-tree) such that the order of the data is still preserved.

#### 3.3.1 Basic Ideas

Before going into details let us first discuss the basic ideas of our approach. Let us consider an MVBT whose page

regions (of the data pages) are depicted in Figure 7. Recall that the page regions are disjoint and that the page regions cover the two-dimensional data space. The partitioning of our example only consists of page regions with at most two temporal predecessor page regions. For example, the temporal predecessor of B is A and the temporal predecessors of H are E and F.

In our approach, the MVBT will be equipped with links to their temporal predecessor, see the example depicted in Figure 7. The reason for using backward-links (and not forward-links) is that backward-links are fully compatible to storing historical nodes on a WORM medium [LS 89]. Backward links point to historical pages which do not change any more. Therefore, there is no additional cost for updating.

Note however that for a write-many medium (e.g. magnetic tape) the MVBT can also use forward-links. Algorithm $Link_{Ref}$ consists of two steps. First, the right border of the query rectangle is used for performing a range-time query. In our example, the answers are in pages D and H. Second, for every qualifying data page obtained in the first step, the temporal predecessor pages are checked whether they can contain an answer. If so, the corresponding pages are read into the buffer, answers are reported and the process is repeated (i.e. the links stored in the predecessor pages are investigated). The example illustrated in Figure 7 is then processed in the following way: First, page D is retrieved and all answers of page D are reported. Thereafter, its predecessor page (B) is examined. The order of the remaining qualifying pages is H, E, F, C. An important feature of $Link_{Ref}$ is that duplicated links will be ignored. In our example, page B is therefore not examined twice. This is achieved by using a method similar to the point-reference method presented in the previous subsection.

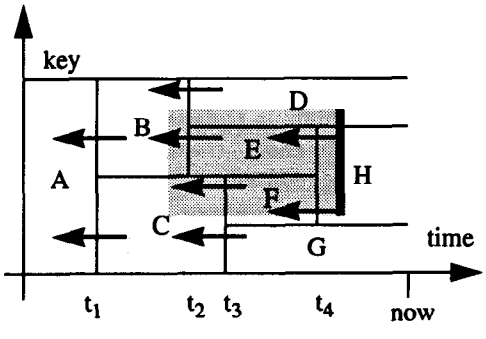



Figure 7: Example of a MVBT and its links

### 3.3.2 The Algorithm

In the following, we present a more formal description of algorithm $Link_{Ref}$. Let A and B be data pages of the MVBT. Page A is a *temporal predecessor page* of B, if the following properties are fulfilled: the insertion time of page B is equal to the (logical) deletion time of A and the key-range of page B intersects with the key-range of page A. It can easily be shown that a data page of the MVBT has at most two temporal predecessor pages.

Algorithm $Link_{Ref}$ exploits the links in a straightforward way. There are only two situations which require some further discussion.

- In order to identify the end of a chain of links without accessing unnecessarily the next predecessor page a link is equipped with some additional information about the region of the predecessor page. If the region of a predecessor page does not intersect with the query rectangle the end of the chain is obviously reached.
- A more difficult problem is how to detect that a link has already been examined. Following a link a second time does not only result in reading the corresponding page once more, but also in traversing the complete chain of links (routed at the duplicate). Therefore, the cost of query processing can considerably be reduced when duplicated links are identified.

Our approach to identifying duplicates is very similar to the one presented in the previous subsection. First of all, recall that a link is only duplicated when a reorganization step consists of a combination of merge & key-split. A reference point of the link can be computed by using the information about the page region of the predecessor page. Let R be the intersection between that page region and the query rectangle. The *reference point* is defined as the lower left corner of R. Thereafter, we test the key of the reference point whether it is in the key-range of the page that contains the link. If so, we follow the link to the temporal predecessor page. Otherwise, the link is identified as a duplicate. An example of the method is given in Figure 8. There are two copies of the link pointing to page A. For the given query rectangle, the reference point is in the key-range of C and therefore, the link stored in C is followed to A.

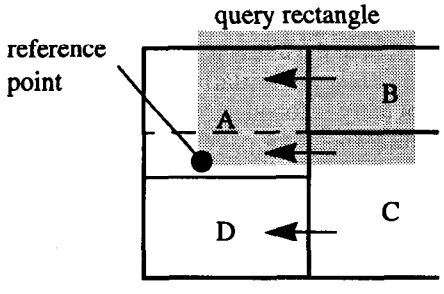



Figure 8: Reference point of a link

The formal description of the algorithm follows:

```
1 Algorithm Link_Ref(Entry f; key_range [R_L ,R_U);
                     time_period [P_B,P_E))
2   N = GetPage(f);
3   compute the page region [K_L,K_U) x [T_L,dummy);
    /* this information is stored in f */
4   if (N is a directory page)
5     for (all entries g in N) do
6       compute the key range [low, up) of g;
7       compute the life-span [t_B,t_E) of g;
```

```
8       if ( [low, up) ∩ [R_L, R_U) ≠ ∅ and
             P_E ∈ [t_B, t_E) )
9           (ref_K,ref_T) = (max(low, R_L), max(t_B, P_E));
10          if ((K_L ≤ ref_K < K_U) and (T_L ≤ ref_T))
11              Link_Ref(g, [R_L ,R_U), [P_B,P_E));
12  else /* N is a data page */
13      for (each entry g in N) do
14          compute the key value k of g;
15          compute the life-span [t_B,t_E) of g;
16          if ( k ∈ [R_L, R_U) and
                 [t_B, t_E) ∩ [P_B, P_E) ≠ ∅ )
17              output g;
18      if (P_B < T_L)
19          for (each temporal predecessor pd of N) do
            /* at most two */
20              compute the key range [low, up) of pd;
21              ref_K = max(low, R_L);
                /* ref_K is the key component of the
                   reference point */
22              if (K_L ≤ ref_K)
23                  Link_Ref(pd, [R_L ,R_U), [P_B,P_E));
24  end; /* Link_Ref */
```

The algorithm $Link_{Ref}$ is initially started with the same parameter setting as used for the initial call of $DF_{Ref}$. The algorithm first reads page N and then computes the page region of N (using f). If N is a directory page, a range-time query is performed using the right border of the two-dimensional query region. This is very similar to the steps of algorithm $DF_{Ref}$. Note that $Link_{Ref}$ recursively calls itself (see line 11).

We first report the answers found in page N. Thereafter, we investigate each of the temporal predecessor pages of page N and compute its key range. Next, the key component of the reference point is computed. If this component is greater or equal than $K_L$, algorithm $Link_{Ref}$ continues in a recursive fashion.

### 3.3.3 Worst-Case Performance Analysis

Let us briefly discuss the worst-case performance of algorithm $Link_{Ref}$. The range-period query consists of a range-time query and a traversal of the data pages. Let us first consider the range-time query. Assume the MVBT contains $n$ versions and that the range-time query is performed on version $v$, $1 \le v \le n$. Let $m_v$ denotes the number of records in version $v$ and let $a_1$ be the number of answers to the range-time query. It was shown in [BGO+ 93] that the number of accesses is

(i) $O(\log_b n + \log_b m_v + a_1/b)$

in the worst-case. Recall that $b$ denotes the capacity of a page. The first term refers to the cost of retrieving the root of version $v$. Let us emphasize that the cost expressed in the other terms is equal to the cost of processing a corresponding range query in an ordinary $B^+$-tree that only stores the records of version $v$.

Now let us consider the traversal of the data pages. The worst-case will appear when no more answers are found, but further pages have to be investigated. This situation can occur when updates (insertions and deletions) were only performed on the pages whose key-range contains the left or right bound of the key range of the query. For a give version (in particular for version $v$), there are at most two of these pages. Since $O(b)$ updates are required to perform a structural change of these pages, the number of disk accesses is then at most

(ii) $O(vp/b)$

where $vp$ denotes the size of the time period of the query (expressed in the number of versions). The worst-case performance of a range-period query is then simply the sum of the formulas (i) and (ii) presented above. Note that when the number of versions can be expressed as a linear function of a1 (the number of answers of the range-time query), the total cost is already given by formula (i).

### 3.3.4 Advanced Techniques for Query Processing

There are two other properties which make algorithm $Link_{Ref}$ very attractive for processing range-period queries:

- $Link_{Ref}$ can easily be extended to report the answers sorted according to time. For that, the algorithm sweeps the key-range from the right border of the query region to the left border. Whenever the sweep-line touch the left point of an interval, the interval will be reported as an answer. In order to be efficient, this require that all pages whose page regions intersect with the sweep-line are kept in main memory. If so, the number of disk accesses would not be affected.
- A modified version of $Link_{Ref}$ is an interesting candidate for processing queries when historical data is stored on magnetic tapes. Let us assume that the technique of the TSBT [LS 89] is used to move historical data from magnetic disk to a tape. Furthermore, let us assume that $Link_{Ref}$ actually uses forward-links (instead of backward-links). A range-period query could now be initiated as a range-time query using the left border of the two-dimensional query region (instead of using the right border). Thereafter, search proceeds forward to the right border such that the page is read next whose deletion point is closest to the left border of the query region. This processing strategy would guarantee that the tape is always moved forward, but never backward.

### 3.3.5 Adaption to other Access Methods

As mentioned previously, links can efficiently be used by MVAMs derived from $B^+$-trees, but not from R-trees. The reason is that the partitioning of an ordinary R-tree allows (arbitrarily high) overlap of its page regions and that empty parts of the data space are not required to be covered by a page region. Therefore, the number of links in such a multiversion R-tree would not be bound as it is the case for the

MVBT. However, let us mention that the link method is applicable to multidimensional access methods that fulfills the following properties: First, there is no overlap between page regions. Second, a point of the dataspace belongs to a page region. The BANG-file [Fre 87], for example, fulfills these properties and therefore, the link method could be used for a multiversion BANG-file.

## 4 Performance Comparison

In this section, we report the results of a preliminary performance comparison of different methods for processing queries on MVAMs. We consider four different depth-first traversal algorithms ($DF_{Pure}$, $DF_{Hash}$, $DF_{Sort}$, $DF_{Ref}$) and the link algorithm ($Link_{Ref}$). Algorithm $DF_{Pure}$ denotes the naive depth-first algorithm which does not eliminate duplicates. All algorithms are implemented on top of the MVBT.

The objective of our set of experiments was twofold. First, we compare the different depth-first traversal algorithms with respect to CPU-time and disk accesses. Second, we show the cost of traversing the directory in a depth-first fashion compared to the cost using the links in data pages. Thus, in these experiments we restrict our discussion only to methods using the reference method ($DF_{Ref}$ and $Link_{Ref}$).

The set of experiments were performed in the following way. In each experiment, a MVBT was created performing 100,000 operations. This results in a MVBT with 100,000 versions. The first 10,000 operations were only insertions, whereas the other ones were a mix of insertions and updates. Experiments with deletions gave similar results than those without deletions. Therefore, these results are omitted from the paper. The parameters p and q (= 1-p) denote the fraction of updates and insertions, respectively. For example, p = 0.9 means that 90% of the operations were updates (not taking into account the first 10,000 insertions). When an insertion is performed, the corresponding record is computed by using a random number generator. An update randomly selects a record from the current version of the MVBT. In this paper, we only report results obtained from experiments (except the last one) with the following parameter settings: bytes per page = 4K, bytes per record = 160, bytes per directory entry = 20. The capacity of a data page and directory page was then 25 records and 200 entries, respectively. For a data page, the lower bound of the weak version condition is therefore 5 (key density), i.e. a page that belongs to a version contains at least 5 records that belong to the same version.

Overall, we created nine MVBTs for p = 0, 0.01, 0.1, 0.25, 0.5, 0.75, 0.9, 0.99, 1.0. There are various parameters of a MVAM which have an impact on the cost of query processing, see [LS 90]. In this study, we only report the results of the fraction of redundant records ($F_{red}$). The parameter $F_{red}$ is defined as the total number of redundant records stored in the MVBT divided by the number of versions. Figure 9 shows $F_{red}$ as a function of p. For p = 0 (insertions only), $F_{red}$ is close to 1.4, i.e. for each record there are 1.4 redundant records on the average. Overall, the values of $F_{red}$ are between 1.3 and 1.7 in our experiments. Since all pages but the current ones are completely filled up the MVBTs occupied about 45 MB of disk storage. Note that the values of $F_{red}$ obtained for the TSBT and WOBT in similar experiments [LS 90] are lower than the ones of the MVBT.

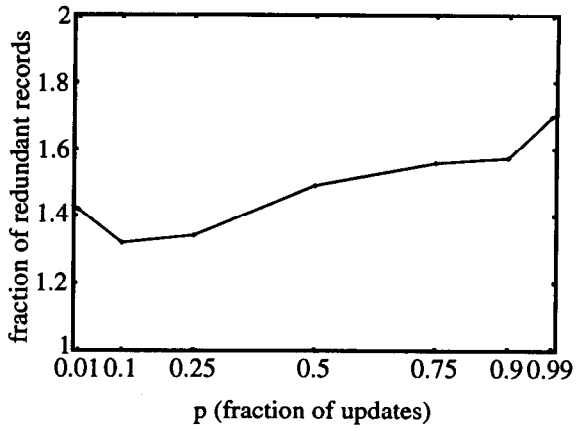



Figure 9: Fraction of redundant records ($F_{red}$)

After the MVBTs had been built we run various sets of queries. The first set ($Q_{key}$) contains 100 key-period queries where the search keys were randomly selected from the corresponding data file and the period includes all versions ([0,...,100000]). The remaining sets of queries only consist of range-period queries where the relative size of the query varied between 0.01% and 1% of the two-dimensional data space. Another important parameter of a query is its *shape factor* (*sf*) which is defined as the relative size of the time period divided by the relative size of the key range. A large shape factor for example means that the key range is rather small compared to the time period. We considered queries with five different shape factors (*sf* = 0.01, 0.1, 1, 10, 100). Each query file $Q_{sf,\ size}$ contains 1000 range-period queries which are described by a constant shape factor *sf* and a constant size *size*.

In our implementations, a query sends its I/O request to a buffer and therefore, the I/O request does not always result in reading the page from disk. The buffer is basically organized according to the LRU replacement policy. In order to keep the current path of the tree in the buffer, the corresponding pages are fixed and unfixed. Before a query is started the buffer pages are invalidated. Thus, the order in which queries are performed does not influence the number of disk accesses required to perform an entire query set.

### 4.1 The Impact of Redundancy

In this subsection, we compare the four different depth-first algorithms: $DF_{Pure}$, $DF_{Hash}$, $DF_{Sort}$, $DF_{Ref}$. Algorithm $DF_{Pure}$ corresponds to the naive depth-first traversal algorithm and produces duplicates (entries and records). Algorithm $DF_{Ref}$ refer to our new method that avoids duplicates by testing the reference point. Algorithm $DF_{Hash}$ uses dou-

ble hashing [Knu 73] to eliminate redundant answers, whereas $DF_{Sort}$ eliminates duplicates by sorting (quicksort). Both algorithms do not avoid duplicated directory entries. $DF_{Hash}$ and $DF_{Sort}$ allocate their memory required for the hash-table and buffer in an optimal fashion, i.e. the algorithm knows about the number of answers and duplicates before the query is actually executed. This admittedly gives advantage to algorithms $DF_{Hash}$ and $DF_{Sort}$.
First of all, we consider algorithm $DF_{Pure}$ and show the impact of redundancy on both the number of duplicates in the response set and the number of disk accesses. This is illustrated using the following two parameters:

- $A_{red} := \frac{\text{number of redundant answers}}{\text{number of answers}}$
- $I/O_{red} := \frac{\text{number of disk accesses}}{\text{number of required pages}} - 1$

The optimum would be when $A_{red}$ and $I/O_{red}$ are equal to 0. For a buffer of 25 pages (100 KB), $A_{red}$ and $I/O_{red}$ are reported for MVBTs with p = 0.5 and query files $Q_{1,\,size}$ where *size* varied between 0.01% and 1%. The graphs show that both parameters increase with an increasing query size. Note that for large query sizes $A_{red}$ can be even higher than $F_{red}$. This is because algorithm $DF_{Pure}$ does not eliminate redundant directory entries. Obviously, this demonstrate the need for eliminating redundant answers.

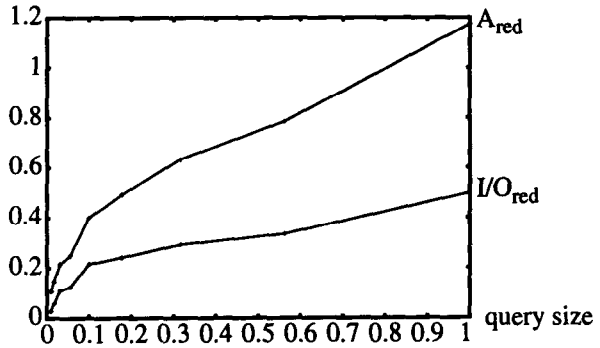



Figure 10: $A_{red}$ and $I/O_{red}$ as a function of the query size (sf=1, p=0.5, buffer capacity=25 pages)

Algorithms $DF_{Hash}$ and $DF_{Sort}$ guarantee that answers are reported only once, but $I/O_{red}$ would be still the same as for $DF_{Pure}$ (because duplicated directory entries are not eliminated). A straightforward solution for reducing $I/O_{red}$ is simply to increase the buffer space. The required space however depends on the size of the database and the query size. For two different query sizes (size=0.32%, 1%), $I/O_{red}$ is depicted as a function of the buffer capacity in Figure 11. The curves show that $I/O_{red}$ declines with an increasing buffer capacity, but only rather slowly in an almost linear fashion.
In contrast to these algorithms, algorithm $DF_{Ref}$ achieves the optimum values for $A_{red}$ and $I/O_{red}$ for arbitrarily large query sizes by keeping only a path of the MVBT in main memory.

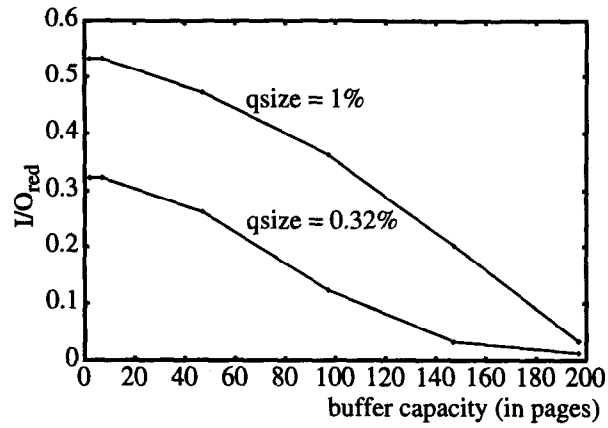



Figure 11: $I/O_{red}$ as a function of the buffer size (sf = 1, p = 0.5)

Let us now consider algorithms $DF_{Hash}$, $DF_{Sort}$ and $DF_{Ref}$ and their cost for eliminating redundant answers. First of all, the first two algorithms require a rather large buffer to eliminate duplicates without causing additional accesses to disks. In our experiments we required a buffer of up to 4 MBytes to eliminate the redundant answers of a query. Note that this is achieved only under the assumption that $DF_{Hash}$ and $DF_{Sort}$ know the exact number of answers and duplicates before the query is executed.

In order to compare the CPU-time of the algorithms, we performed the same experiment as before (MVBT with p = 0.5 and query files $Q_{1,\,size}$, $0.01\,\% \le size \le 1\,\%$). We run these experiments on a IBM P43-133 with 64 MBytes main memory and measured the CPU-time consumption of the different algorithms for performing the queries of $Q_{1,\,size}$, $0.01\,\% \le size \le 1\,\%$. Note that the available main memory was large enough to keep the entire MVBT and therefore, paging did not influence our numbers at all. The results of our experiments are plotted as a function of the query size in Figure 12. For comparison reasons, we also provide the

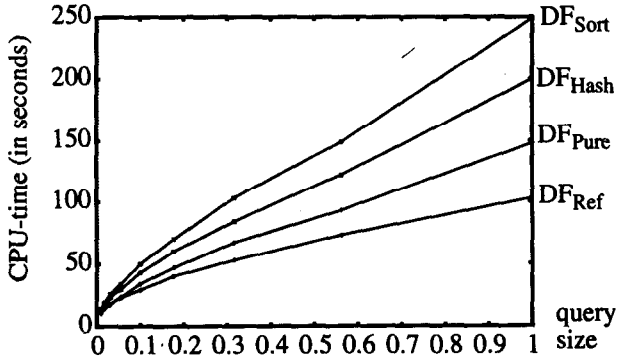



Figure 12: CPU-time as a function of the query size (sf=1, p=0.5)

CPU-times for algorithm $DF_{Pure}$. The graph shows that $DF_{Sort}$ requires most CPU-time among the different algo-

rithms. The cost of $DF_{Hash}$ is between the cost of $DF_{Pure}$ and $DF_{Sort}$. At a first glance, it is surprising that $DF_{Ref}$ is superior to $DF_{Pure}$ (although $DF_{Ref}$ needs additional computations to test the reference point). The reason is that $DF_{Ref}$ visits a page only once, whereas $DF_{Pure}$ can visit a page more than once.

Overall, the experiments show that $DF_{Ref}$ is superior to the other algorithms according to CPU-time and disk accesses. Moreover, $DF_{Ref}$ requires only a few buffer pages (to keep the current path of the MVBT), whereas the required memory of $DF_{Hash}$ and $DF_{Sort}$ growth linear in the number of answers. Since very large queries frequently occur in a transaction-time database, these algorithms face the risk that main memory is too small.

### 4.2 A Comparison between the Link Algorithm and the Depth-First Algorithm

The next sets of experiments were intended to compare the query performance of the link algorithm ($Link_{Ref}$) and the one of the depth-first algorithm ($DF_{Ref}$). From the performance point of view, algorithm $Link_{Ref}$ requires less disk accesses to directory pages than algorithm $DF_{Ref}$. There is no performance gain according to the number of data pages. On the contrary, the capacity of a data page can be slightly less for the link method since (at most) two links have to be stored in the pages additionally. This difference is however not very significant and we assume that the capacity of data pages is the same for both algorithms..

Table 1: Number of disk accesses for query file $Q_{key}$

| p | $Dir_{I/O}$ ($DF_{Ref}$) | $Dir_{I/O}$ ($Link_{Ref}$) | $Data_{I/O}$ | #Answers |
|---|---|---|---|---|
| 1 | 27.38 | 2 | 25.96 | 10.27 |
| 0.99 | 18.98 | 2 | 25.53 | 9.49 |
| 0.9 | 24.21 | 2 | 20.62 | 6.92 |
| 0.75 | 18.65 | 2 | 17.7 | 4.46 |
| 0.5 | 15.91 | 2 | 15.09 | 2.71 |
| 0.25 | 14.04 | 2 | 12.17 | 1.38 |
| 0.1 | 14.67 | 2 | 13.26 | 1.36 |
| 0.01 | 14.8 | 2 | 13.28 | 1.03 |
| 0 | 14.75 | 2 | 13.3 | 1 |

The greatest difference in performance between algorithm $Link_{Ref}$ and algorithm $DF_{Ref}$ occurs when key-period queries are performed. For query file $Q_{key}$, the average number of disk accesses are reported in Table 1. The second and third column show the number of accesses to directory pages using algorithm $DF_{Ref}$ and $Link_{Ref}$, respectively. The fourth column gives the number of accesses to data pages, whereas the number of answers are reported in the last column. Algorithm $Link_{Ref}$ requires only two directory accesses which is equal to the height of the tree, whereas algorithm $DF_{Ref}$ requires significantly more directory accesses. Moreover, algorithm $DF_{Ref}$ retrieves more directory pages than data pages, whereas the performance of the link algorithm is only determined by the number of accesses to data pages.

For range-period queries, we run several other experiments. The results showed that algorithm $Link_{Ref}$ is noticeably superior to $DF_{Ref}$ only for range-period queries where the specified period is small compared to the specified range (large shape factor) and where the capacity of a directory page does not largely differ from the capacity of a data page. In order to demonstrate the performance advantage of algorithm $Link_{Ref}$ for such a setting, we performed several experiments where the capacity of a directory page was set to 25 (= the capacity of a data page). This results in MVBTs where the directory is considerably larger than for those MVBTs generated in our previous experiments. For p = 0.1, 0.5, 0.9, the results of processing query files $Q_{sf,\ 0.01\%}$ are depicted in Figure 13 as a logarithmic function of the shape factor *sf*. For a small shape factor, the performance gain of $Link_{Ref}$ is marginal. With an increasing shape factor, however, $Link_{Ref}$ performs considerably better than $DF_{Ref}$. For *sf* = 100, $Link_{Ref}$ requires only about 60% of the accesses of $DF_{Ref}$. This results indicate that for a high shape factor $DF_{Ref}$ requires most of the disk accesses for retrieving directory pages.

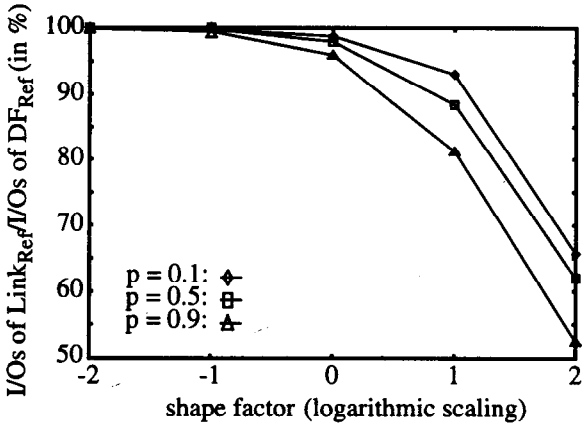



Figure 13: Number of disk accesses of $Link_{Ref}$ compared $DF_{Ref}$ (*size* = 0.01%)

## 5 Conclusions

In this paper we studied query processing techniques for transaction-time databases where data is never physically deleted. Due to the size of a transaction-time database, access methods are inevitably required to efficiently support queries. An important type of query is the range-time query where records are retrieved from the database whose keys are in a given range during a given time period. We investigated how to evaluate range-period queries efficiently using multi-version access methods (MVAMs).

One of the most efficient MVAMs is the multiversion B-tree (MVBT) [BGO+ 93]. The MVBT optimally solves the range-time query problem. Therefore, we primarily

restricted our discussion to the MVBT and related MVAMs ([Eas 86], [LS 89]). These MVAMs occasionally duplicate records and index entries to separate data according to time. Traditional algorithms for processing range-period queries are not aware of these duplicates and therefore, duplicates also occur in the response set of a query. Moreover, the performance of these algorithms can considerably decline because of the duplicates in the directory.

We presented two new algorithms for processing range-period queries without reporting duplicated answers. The one algorithm traverses the MVBT in a depth-first order from a root to the leaves. For each qualifying entry and record, a reference point is computed which identifies duplicated entries and records. The other algorithm can be viewed as a generalization of the classical algorithm for performing range queries in $B^+$-trees. The data pages of the MVBT are linked together and these links are exploited to retrieve the data pages which are required for evaluating the query. In contrast to $B^+$-trees, links can also be duplicated. Therefore, we also equipped the link algorithm with an efficient technique for avoiding duplicates. Both algorithms achieved that duplicates are not reported almost without inducing any additional cost.

We compared the performance of our algorithms with depth-first algorithms that eliminated duplicates using traditional methods (hashing, sorting). Results of an experimental comparison showed that these algorithms were constantly inferior to our algorithms according to disk accesses and CPU-time although our algorithms used considerably less buffer space.

In our future work, we will investigate how to organize historical data on tertiary storage such as tapes and magneto-optical disks. Moreover, we are also interested in extending our work to bitemporal access methods [KTF 95].

**Acknowledgement**

We are very thankful to Beatrice Bott for providing a first implementation of the MVBT.

**References**

[BFA 86] R. Barrera, A. Frank, K. Al-Taha. Temporal Relations in Geographic Information Systems: A Workshop at the University of Maine. *SIGMOD Record*, 1991, 20:85-91.

[BGO+ 93] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer. On Optimal Multiversion Access Structures. *Proc. Symp. on Large Spatial Databases*, in Lecture Notes in Computer Science, Vol. 692, pp 123-141, Singapore 1993.
An extended version will appear in the VLDB Journal.

[Bra 84] K. Bratbergsengen. Hashing methods and relational algebra operations. *Proc. VLDB*, 1984, pp. 323-333.

[BS 96] J. van den Bercken, B. Seeger. Query Processing Techniques for Multiversion Access Methods. Technical Report, Nr.11, Department of Mathematics, Philipps-University, Marburg, 1996

[CCS 93] E. F. Codd, S. B. Codd, C. T. Salley. Providing OLAP to User-Analysts: An IT Mandate. *Arbor Software White Papers*, available via http://www.arborsoft.com/papers.

[Eas 86] M. Easton. Key-sequence data sets on inedible storage. *IBM J. of Research and Development*, 1986, 30:230-241.

[Fre 87] M. Freeston. The BANG-file: A New Kind of Grid File. *ACM SIGMOD*, 1987, pp. 260-269.

[Gut 84] A. Guttman. R-trees: a dynamic index struture for spatial searching. *Proc. ACM SIGMOD*, 1984, 47-57.

[Knu 73] D. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

[KS 91] C. Kolovson, M. Stonebraker. Segment Indexes: dynamic indexing techniques for multi-dimensional interval data. *Proc. ACM SIGMOD*, 1991, 138-147.

[KTF 95] A. Kumar, V. Tsotras, C. Faloutsos. Access Methods for Bi-Temporal Databases. *Proc. Workshop on Temporal Databases*, Zürich, 1995, pp. 235-254.

[LS 89] D. Lomet, B. Salzberg. Access Methods for Multiversion Data. *Proc. ACM SIGMOD*, 1989, pp. 315-324.

[LS 90] D. Lomet, B. Salzberg. The Performance of a Multiversion Access Method. *Proc. ACM SIGMOD*, 1990, pp. 353-363.

[MKW 96] P. Muth, A. Kreiß, G. Weikum. LoT: Dynamic Declustering of TSB-Tree Nodes for Parallel Access to Temporal Data. *Proc. 5th Int. Conf. on Extending Database Technology*, 1996.

[SA 85] R. Snodgrass, I. Ahn. A Taxonomy of Time in Databases. *Proc. ACM SIGMOD*, 1985, pp. 236-246.

[SK 88] B. Seeger, H.-P. Kriegel. Techniques for Design and Implementation of Spatial Access Methods. *Proc. VLDB*, 1988, pp. 360-371.

[SRF 87] T. Sellis, N. Roussopoulos, C. Faloutsos. The $R^+$-tree: a dynamic index for multi-dimensional objects. *Proc. VLDB*, 1987, pp. 507-518.

[ST 94] B. Salzberg, V. Tsotras. A Comparison of Access Methods for Time Evolving Data. *Technical Report CATT-TR-94-81*, 1994, Polytechnic University, Brooklyn, NY.

[Sto 87] M. Stonebraker. The Design of the POSTGRES Storage System. *Proc. VLDB*, 1987, pp. 289-300.